Suplemento mensual dedicado a la imagen sintética. Número 18

Render MANIA

Informe Especial. Paisajes estelares con Galaxy.inc

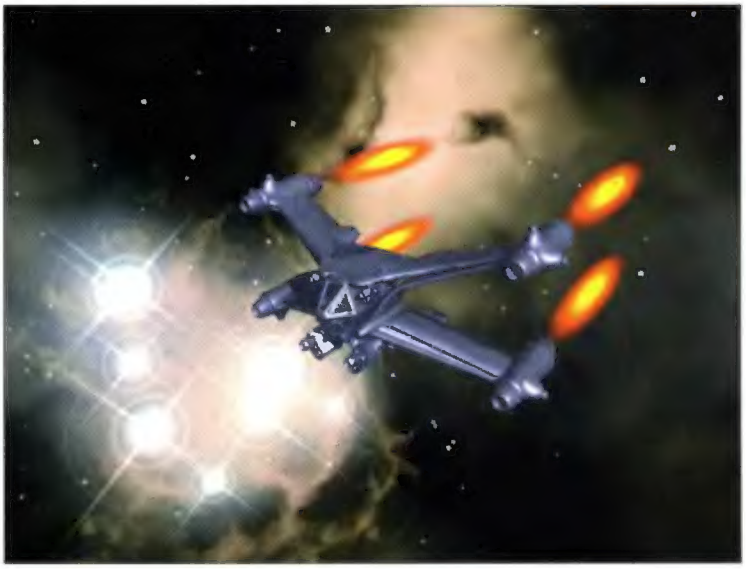
Cómo... Crear texturas propias con POV

Foro del Lector. Belleza sintética

pcmania

José Manuel Muñoz

Funcionan las texturas procedurales de POV (parte I)



Este modelo es una versión libre del caza Starfury de Babylon 5. Ha sido creado por Mark Kane y fue publicado en Pcmanía número 11.

En el número anterior incluimos un pequeño resumen sobre el tema de las texturas para que los Povmaníacos recién enganchados pudiesen utilizar las texturas incluidas en la librerías de «POV». En esta ocasión, vamos a profundizar un poco más en este tema, de manera que nuestros lectores más recientes puedan crear sus propias texturas procedurales.

Los creadores de «POV» definen, en su libro "Raytracing II", la palabra textura como "...la descripción del material con el que está hecha una figura..." Y también indican que "Las texturas sólo afectan al aspecto superficial de los objetos y figuras y nunca a su forma real". *(Aunque este libro publicado por Anaya trata de la versión 2.2 de «POV», los capítulos dedicados al tema de las texturas siguen siendo –al igual que la parte de referencia– muy útiles aunque no contemplan las novedades de la versión 3.0).* Esto no impide que algunas texturas parezcan alterar completamente la forma de los objetos. Como veremos a continuación, el lenguaje escénico de «POV» dispone de un conjunto muy completo de sentencias para definir texturas. Con estas sentencias podremos definir el color de las superficies, la forma en que se ha de reflejar la luz, la transparencia, la rugosidad, etc. Así, definiendo valores para todas estas propiedades, podremos crear nuestras propias texturas para simular agua, metales, madera, piedra, etc.

Bloques de comandos

La definición completa de una textura se realiza en tres apartados básicos donde se definen todos los aspectos de la misma. Con las sentencias

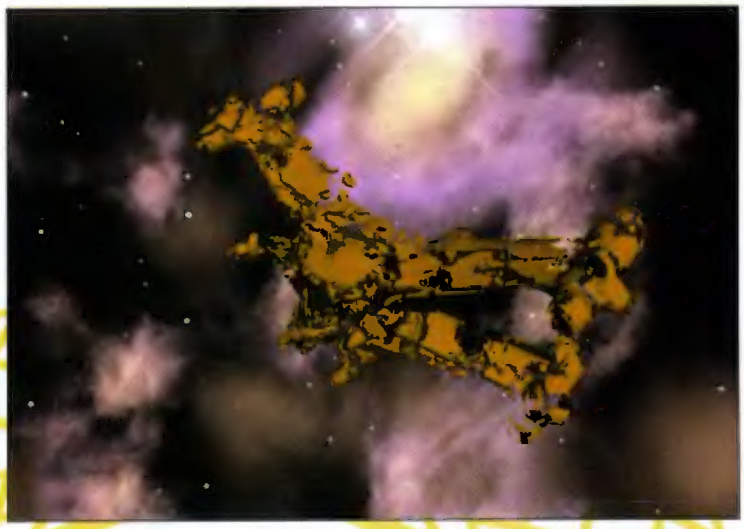
englobadas dentro del apartado "pigment" definiremos el color o el mapa de colores de la textura. Con las sentencias incluidas dentro del apartado "normal" quedará definida la rugosidad (o falta de ella) de la superficie del material, y con las sentencias pertenecientes al apartado finish declararemos algunas propiedades relativas al acabado del material (grado de transparencia, brillo, porcentaje de reflexión, etc.). Así, la definición básica de una textura queda como sigue:

```
texture{
  pigment{ lista de sentencias }
  normal{ lista de sentencias }
  finish{ lista de sentencias }
}
```

Esta división en tres bloques de las sentencias utilizadas para crear texturas no obedece tan solo a criterios de claridad sino también a diversas razones prácticas. Una de ellas es que «POV» nos permite aprovechar definiciones anteriores para crear variables, objetos o texturas nuevas. Esta propiedad es la que hace posible definir un bloque pigment y emplearlo posteriormente en la definición de varias nuevas texturas. Tenemos un claro ejemplo de esto en el fichero metals.inc, del directorio include, donde se definen diversos bloques finish con diferentes tipos de acabado que posteriormente son empleados en la definición de texturas metálicas. Si echáis un vistazo a dicho fichero observaréis muchas líneas parecidas a la siguiente:

```
#declare T_Brass_1A = texture { pigment {
P_Brass1 } finish { F_MetalA  } }
```

En esta línea se está definiendo una nueva textura aprovechando dos bloques ya definidos anteriormente: el pigmento P_Brass1 y el acabado F_MetalA. Al ver esta línea quizá os preguntéis dónde esta el apartado "normal" y la respuesta es que, al no haberlo escrito nosotros, «POV» ha incluido el bloque "normal" definido por defecto. Lo cierto es que «POV» crea valores por defecto para todas las sentencias que se emplean para crear texturas. Esto nos permite crear nuevas texturas escribiendo solamente las sentencias cuyos valores se van a desviar de los parámetros dados por defecto. Lo cual está muy bien porque hay más de 40 sentencias pensadas para crear texturas, y sería un verdadero incordio escribirlas todas en la definición de cada nueva textura.



En esta imagen podemos ver un experimento interesante, una textura diseñada para destrozar objetos.

Una variación sobre este tema radica en la posibilidad de definir nuevas texturas tomando como base otras texturas completas definidas anteriormente. Cuando sea este el caso deberemos escribir en primer lugar el identificador de la textura que nos va a servir como base y añadir después los apartados que sean convenientes. Veamos un ejemplo:

```
#declare mia=texture{ T_Brass_1 finish{
ambient 0.10 }}
```

En este ejemplo la nueva textura llamada mia empleará los mismos bloques pigment y normal utilizados en la textura T_Brass_1 anteriormente definida. Incluso empleará las sentencias utilizadas en el bloque finish F_MetalA que se incluyó en T_Brass_1. La única diferencia será que el valor dado a ambient (una de las sentencias que pueden usarse dentro del apartado finish) será el declarado en mia y no el utilizado por T_Brass_1.

Para que comprobéis esto hemos escrito un corto fichero escénico llamado tprueba1.pov que hallaréis en el CD, en Como68. En él hay definidas dos texturas que emplean el acabado metálico F_MetalA. La primera (tprueba1) incluye este acabado directamente en su definición y la segunda (tprueba2) lo tiene porque ha incluido a la textura anterior como base de su propia definición.

En la escena obtenida al renderizar este fichero veremos dos esferas: en la de la izquierda hemos aplicado tprueba1 y en la segunda tprueba2. Como veréis, la esfera de la derecha muestra

Figura 1: "textura de la esfera de la deracha"

```
    ...
    ...
#declare mimapa2=color_map {
  [0.0    color rgb <1, 0, 0>]
  [0.5    color rgb <1, 1, 0>]
  [1.0    color rgb <1, 0, 0>]
}
#declare tprueba4=texture{pigment{wood color_map{mimapa2}}}
    ...
    ...
```

también un acabado metálico similar al de la otra, pero exhibiendo unas tonalidades más oscuras que se deben al cambio realizado en el valor ambient del apartado finish.

Ahora veamos otro caso. Supongamos que realizamos un par de cambios en el ejemplo anterior. Añadiremos la siguiente línea antes de la declaración de la textura tprueba2:

```
#declare acabado1=finish{ambient 0.10}
```

y luego cambiaremos un poco la declaración de tprueba2:

```
#declare tprueba2=texture{tprueba1 finish{
acabado1}}
```

Al ver esto probablemente muchos pensaréis que obtendremos el mismo resultado anterior al renderizar. Pues bien, esto no será así sino que la esfera con la textura tprueba2 perderá su aspecto metálico. Esto se debe a que al incluir el identificador "acabado1" dentro del bloque finish, todo el bloque finish definido para acabado1 sustituye al bloque finish de tprueba1. El nuevo acabado de la esfera es distinto al anterior porque en el bloque acabado1 POV ha añadido los valores que hay definidos por defecto para las sentencias reservadas del apartado finish. Como estos valores por defecto no coinciden con los definidos para el bloque finish F_MetalA (en el fichero metals.inc), el resultado es diferente.

Aplicando algo de turbulencia a la trama wood.

Resumiendo: al incluir un identificador dentro de un apartado pigment, normal o finish, se emplearán todas las propiedades definidas anteriormente para ese identificador. (Naturalmente el tipo del identificador habrá de coincidir con el del apartado donde lo escribamos. No podemos meter un identificador definido como pigment dentro de un bloque finish). Si, por el contrario, no ponemos ningún identificador y añadimos una sentencia nueva dentro de uno de los bloques de una textura ya definida, entonces la nueva propiedad se sumará a las que ya tuviera definidas la textura en dicho bloque.

Por último, debéis recordar siempre la ley de los bloques contenedores. Por ejemplo, en el caso siguiente:

```
union{object{cabeza}
      object{nariz}
      object{ojo texture{tojo}}
      texture{colorpiel}
}
```

la textura colorpiel se aplica en este caso sobre los objetos de la unión "cabeza" y "nariz" mientras que la textura tojo se aplica sólo sobre el objeto "ojo", que ignora a colorpiel al tener incluida su propia textura. Si en este ejemplo hubiésemos puesto también texturas dentro de los bloques de "cabeza" y "nariz", entonces colorpiel no se aplicaría sobre ningún objeto y su

Un experimento caprichoso cambiando el color normal asignado en la trama de colores del caza.

inclusión sería superflua, aunque no generaría error alguno.

Pigment

Como ya hemos visto, la sentencia pigment se emplea para abrir un bloque donde escribiremos las sentencias que definen el color o la trama de colores de la textura. Hay que recordar que pigment define el color propio de las superficies, el cual no es necesariamente el que veremos siempre al renderizar la escena.

Si por ejemplo tenemos un objeto con un pigmento blanco e iluminamos la escena con una luz azul, el resultado será un objeto de apariencia azulada. Como ya vimos en números anteriores podemos escribir cosas como:

```
pigment{ White }
```

donde empleamos un color definido en el fichero colors.inc. Y también podemos especificar un color directamente utilizando la sentencia RGB:

```
pigment{rgb<1,1,1>}
```

Todo esto nos servirá para especificar colores lisos, pero, por citar sólo dos casos, ¿cómo se generan las tramas de colores que pueden verse en las texturas de madera y mármol? La respuesta es compleja y quizá por ello muchos povmaníaco no han profundizado en este tema, limitándose durante mucho tiempo a utilizar las excelentes texturas que vienen definidas en los archivos include de «POV» (como woods.inc, stones1.inc, textures.inc y otros). Sin embargo, si queremos crear nuestras propias texturas o al menos manipular las ya existentes, estamos obligados a saber cómo funciona esto.

Para empezar «POV» dispone de una larga serie de sentencias como wood,

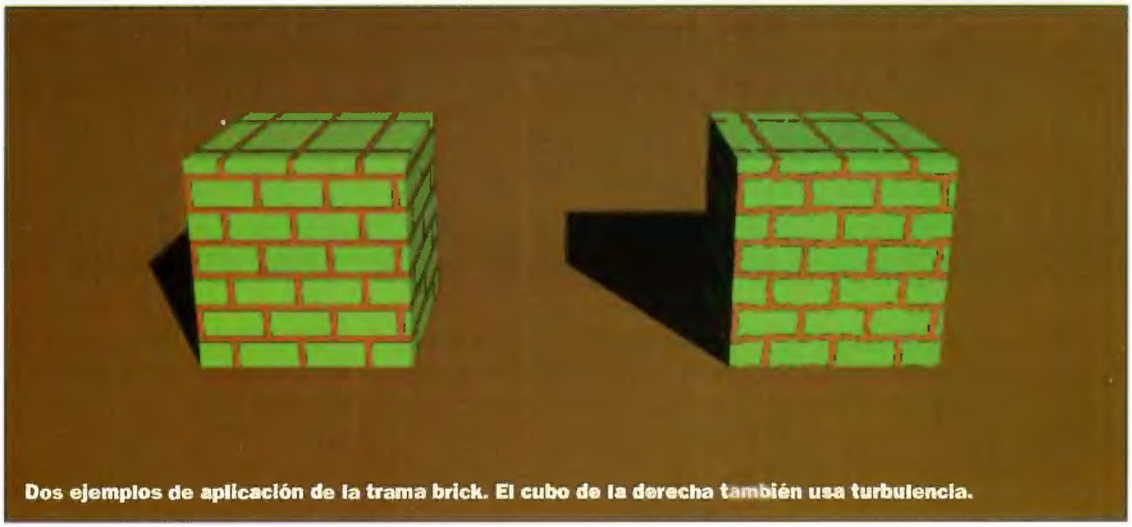
Figura 2: "textura de la esfera de la derecha"

```
      ...
      ...
#declare mimapa1=color_map {
   [0.0    color rgb <1, 0, 0>]
   [0.5    color rgb <1, 0, 0>]
   [0.5    color rgb <1, 1, 0>]
   [1.0    color rgb <1, 1, 0>]
}
#declare tprueba3=texture{pigment{wood color_map{mimapa1}}}
      ...
      ...
```

granite, marble y muchas otras más que generan tramas internas que se utilizan para alterar el color de las superficies. Estas tramas o patrones asignan valores –que oscilan entre 0.0 y 1– a cada punto 3D de las superficies donde se aplican. Y con dichos valores se asigna a cada punto 3D un color tomado de una tabla de referencia de colores. El usuario puede definir su propia tabla de colores utlizando la sentencia color_map o bien puede emplear la tabla por defecto. Estas tablas o mapas de colores definen una lista de intervalos a los que se asocian colores. Para entender esto un poco mejor, examinad el fichero de ejemplo tprueba2.pov. En él hemos creado un par de texturas que usan la sentencia wood e incluyen un mapa de colores propio. Estas dos texturas se aplican sobre dos esferas. En la de la derecha hemos aplicado la textura que podéis ver en la Figura 1.

En esta figura podéis ver la estructura que se emplea para definir un mapa de colores. Este mapa se encierra entre llaves y se define con una lista de referencias que debe cubrir los valores de intensidad de 0.0 a 1. Cada referencia se encierra entre corchetes y dentro de cada una se especifican primero el valor de intensidad y luego el color que deberá mapearse sobre las superficies en los puntos donde la trama tenga la intensidad de la referencia. En nuestro ejemplo el color para la intensidad 0 será el rojo. Posteriormente, el color será amarillo en los puntos de intensidad 0.5 y volverá a ser rojo en los puntos de intensidad 1.0. Seguramente ya os estaréis preguntando qué color se mapeará en los puntos de intensidad intermedia entre dos referencias. Y la respuesta está en que el color de esos puntos se calculará por interpolación lineal de las dos referencias entre las que se hallen comprendidos. Así, por ejemplo, el color de un punto de intensidad 0.25, estará a medio camino entre el rojo y el amarillo.

En cuanto a la palabra wood, crea una trama de intensidades que forma anillos concéntricos paralelos al eje Z. Cuando esta trama se agita un poco y el mapa de colores ha sido adecuadamente definido (lo cual no es el caso de nuestro ejemplo), el resultado es similar al de los anillos concéntricos que pueden verse en la madera real. Por defecto, cada anillo está separado del siguiente por una unidad de distancia, por lo que tendremos que emplear el comando scale para ajustar el tamaño en objetos cuyas dimensiones sean demasiado grandes o demasiado pequeñas para la trama.



Dos ejemplos de aplicación de la trama brick. El cubo de la derecha también usa turbulencia.

Por otro lado, puede ocurrir que por alguna razón nos interese que haya zonas de color liso como las que podéis ver en la otra esfera del ejemplo anterior, donde el color de los anillos pasa bruscamente de rojo a amarillo y viceversa. Esto podemos conseguirlo utilizando un mapa de colores como el empleado en dicha esfera y que podéis ver en la Figura 2.

Para conseguir una zona lisa entre dos rangos de intensidad basta con in-

dicar el mismo color en las dos referencias implicadas, con lo cual no se produce interpolación.

Ahora avancemos un poco más en este tema. Hemos dicho que wood genera una trama que suele emplearse para crear texturas de madera pero, como acabamos de ver, los resultados que se obtienen con ella son excesivamente regulares. Hace falta algo más y la respuesta está en las sentencias que «POV» ofrece para desorganizar y agitar las tramas.

Turbulence

Esta sentencia se emplea para indicar el grado de agitación que debe emplearse sobre una trama aplicada a una superficie. Turbulence puede emplear como parámetro un valor en flotante o un vector y los valores admitidos van desde 0 (no hay ninguna turbulencia) a 1 (máxima turbulencia). Cuando se emplee un vector podremos especificar diferentes grados de turbulencia para cada eje espacial (no olvidéis que las tramas son tridimensionales y afectan a todo el volumen espacial del universo de «POV»). Así, por ejemplo, la sentencia:

```
turbulence <0.2, 1, 0>
```

genera algo de turbulencia a lo largo del eje X, un grado máximo de turbulencia en el eje Y, y ninguna turbulencia en el eje Z. (Nota: la sentencia turbulence se emplea para muchas cosas más aparte de para agitar las tramas de pigmentación. Con turbulence podemos también agitar las normales de una superficie o añadir turbulencia a los halos o a la niebla). Si renderizáis el archivo de ejemplo trprueba4.pov podréis comprobar cómo afecta la turbulencia a las texturas wood explicadas en tprueba2.

Turbulence emplea una función de ruido aleatorio llamada Dnoise que genera una agitación de la trama siguiendo un número determinado de pasos que se indican con la sentencia "octaves". El valor, por defecto, para octaves es de 6 y el tiempo de cálculo va aumentando progresivamente con valores mayores. (Nota: el rango de valores permitido para octaves va de 1 a 10). La turbulencia va decreciendo paulatinamente con cada nuevo paso u octava. Este decrecimiento se indica con la sentencia omega cuyo valor por defecto es de 0.5 (lo cual significa que cada nuevo paso tiene la mitad del tamaño del anterior). En cuanto al grado de aleatoriedad relativa entre cada octava, éste se aplica según el valor que demos como parámetro a la sentencia lambda cuyo valor por defecto es de 2.0, lo que produce unos resultados bastante aleatorios.

Las propiedades de la textura de la esfera derecha han sido "heredadas" de la textura aplicada en la esfera izquierda.

Nota: aunque predecir los resultados que se obtendrán con los valores dados a turbulence no es excesivamente difícil, sí que puede serlo intentar hacer lo propio con los valores dados a las sentencias octaves, omega y lambda. Aquí, como casi siempre, dependeremos de los experimentos propios para conseguir el resultado deseado.

Tipos de tramas

Seguidamente vamos a incluir una pequeña lista donde se describen los patrones permitidos por «POV». Todas las tramas que generan estos patrones se aplican en el plano X-Y, a lo largo del eje Z y tienen su centro en las coordenadas <0,0,0>. (Nota importante: por eso el anillo más interno producido por wood arranca de esa posición. Si queremos que dicho anillo resulte visible en objetos situados en otras posiciones tendremos que incluir una orden translate al final del bloque texture. Otra posibilidad es crear el objeto en la posición donde la aplicación de la textura nos resulta interesante y luego utilizar una orden translate al final del bloque del objeto). En cuanto a las sentencias turbulence, octaves, omega y lambda suelen utilizarse para alterar –aunque hay excepciones– las tramas que generan las sentencias que detallamos a continuación:

• Agate: Con esta sentencia podremos crear tramas de pigmentación parecidas a las que pueden lograrse con marble, turbulenta y con manchas. Este

patrón emplea una función de turbulencia propia cuya fuerza puede controlarse asignando un valor en flotante a la sentencia agate_turb. La sentencia turbulence también puede ser empleada (como octaves, omega y lambda), aunque Agate ya es lo suficientemente turbulenta de por sí.

• Bozo: Este patrón suele emplearse para crear nubes (añadiéndole un poco de turbulencia) ya que usa una función de ruido que genera unos resultados muy suaves gracias a que los puntos cercanos tienen valores de intensidad similares.

• Brick: Esta sentencia sirve para generar un patrón de ladrillos. Su uso requiere el empleo de algunas sentencias adicionales y su sintaxis es:

```
pigment {
  brick color_1, color_2
  brick_size vector
  mortar flotante
}
```

Aquí, color_1 indica el color del mortero que rodea a los ladrillos mientras que color_2 especifica el color de los propios ladrillos. Brick_size sirve para indicar las dimensiones de cada ladrillo. Los valores dados en cada eje se usan para especificar la longitud de cada ladrillo a lo largo de ese mismo eje. El valor flotante que sigue a mortar indica el grosor de la capa de mortero que separa a los ladrillos -habrá problemas si, por despiste, damos a esta capa un grosor superior al de cualquier longitud dada a los ladrillos-. La trama brick no emplea color_map aunque sí turbulence. Hallaréis un ejemplo en tprueba5.pov.

• Bumps: Esta sentencia, ideada para ser utilizada sólo desde el apartado normal, genera una trama similar a la de las sentencias Bozo y spotted cuando se la utiliza desde el bloque de pigment.

• Checker: Produce un patrón a cuadros alternando dos colores. Suele ser empleado en tableros de ajedrez o en suelos virtuales creados para comprobar la corrección de las dimensiones de los objetos. Su sintaxis es:

```
pigment{ checker pigment{color_1}, pigment
{color_2}}
```

Se necesita especificar dos colores para los cuadros que van a repetirse. Por defecto, la longitud de los lados de los cuadros es de una unidad. Realmente no se trata de cuadros, sino de cubos ya que la textura se extiende también por el eje Z.

• Crackle: Este patrón es interesante ya que genera una trama de polígonos (en una imagen de tipo tile, cada lado de la misma encaja con el lado opuesto). Esto puede ser útil para crear suelos o paredes con piedras irregulares (como las de un castillo). Afortunadamente crakle puede usar color_map.

• Dents: Este patrón, como bumps, fue originalmente diseñado para ser empleado sólo en el apartado normal. (No sabemos cómo describir este patrón pero parece adecuado para texturas metálicas y para crear texturas que simulen superficies acuáticas o arenosas).

• Gradient: Genera una trama de colores formada por capas que se extienden en la dirección del eje que se indica en el vector que le acompaña. Su sintaxis es:

```
pigment{gradient vector}
```

Por ejemplo, la línea

```
plane {y,0 pigment{ gradient
<0,0,1> scale<10,10,10>}}
```

crea un plano de bandas de líneas horizontales que se extienden hasta el horizonte.

• Granite: Esta sentencia emplea una función fractal de ruido para generar un buen patrón de granito. Este tipo de trama es muy utilizada por los ficheros de tipo stones.inc para crear texturas de piedra. El buen acabado de estas texturas depende también del empleo de un buen mapa de colores y del uso de las sentencias de distorsión de patrones.

• Hexagon: Esta sentencia es similar a checker pero con la diferencia de que genera prismas hexagonales en lugar de cubos. Por esta razón se requieren tres colores con lo que su sintaxis es:

```
pigment{ hexagon color_1, color_2, color_3}
```

Pruebas de la trama de granito.

Continuará

Todavía nos queda por ver una buena lista de patrones y también tenemos que describir el uso de los bloques normal y finish. Y también, por supuesto, crearemos algunas texturas procedurales que sirvan como ejemplo.

Galaxy.inc: un POV-IPAS para crear cuerpos celestes

Presentamos un nuevo POV-IPAS capaz de generar magníficos paisajes estelares. Con Galaxy.inc de Chris Colefax –uno de los mejores creadores de efectos para «POV»– podremos crear fácilmente bellas escenas celestes en las que aparecerán Galaxias, nebulosas, agrupaciones de estrellas, cometas y meteoros.

En el número 3 de Rendermanía presentamos un método para crear paisajes estelares utilizando los halos de «POV», que ya fueron estudiados en el número 2. La técnica empleada era muy sencilla: se creaban uno o dos halos para generar nebulosas de fondo y luego se empleaban otro tipo de halos para crear las estrellas más grandes. Esta técnica, sin embargo, no estaba exenta

de problemas, debidos a que el cúmulo estelar se creaba aleatoriamente dentro de un volumen espacial que también era compartido por los halos de las nebulosas.

Esto a veces provocaba como resultado unos fallos visuales cuando los halos de las estrellas interferían con los de las nebulosas. La solución más simple hubiera sido, por supuesto, emplear varios planos espaciales para los objetos. Podríamos, por ejemplo, haber reservado el plano más cercano a la cámara para los objetos más próximos como las naves espaciales y los campos de asteroides. Posteriormente, en orden de lejanía creciente, se podrían haber reservado volúmenes espaciales para las estrellas más cercanas, las nebulosas, las estrellas más lejanas, etc. Todo esto no llegó a ponerse en práctica pero previsiblemente también habría implicado problemas de realización, probablemente de velocidad de render.

Aquí teneis todos los objetos estelares que ha creado Colefax.

En definitiva, quizá algún día escribamos algún otro POV-IPAS (o include a secas, como empieza a llamarse a estos archivos en la red) para comprobar estas teorías. Por ahora nos limitaremos a estudiar el paquete Galaxy.inc, cuya última versión publicó Colefax en febrero de este año y que podéis hallar en el CD-ROM.

Instalación

Tenemos dos opciones de instalación: podemos descomprimir el ZIP que contiene al POV-IPAS dentro del directorio donde vayamos a crear las escenas o bien podemos hacer esto dentro del directorio include de nuestra instalación de «POV». Si optamos por esta última opción podremos emplear los ficheros de Colefax desde cualquier directorio, simplemente indicando la ruta al directorio include con la opción de la línea de comandos "L" que ya explicamos en el número anterior.

La filosofía de Galaxy.inc

La filosofía del paquete de Colefax para generar paisajes estelares se basa en una idea radicalmente distinta a la que ya comentamos en el número 3. Sorprendentemente, en lugar de utilizar halos, Colefax emplea simples discos sobre los que genera diferentes texturas empleando tramas de pigmentación. Por supuesto, el tipo de trama empleada (granite, onion y otras) dependerá –al igual que los mapas de color usados– de cuáles sean los objetos celestes cuya apariencia se desea simular. (Nota: hay algunas excepciones. El fondo de estrellas lejanas se crea sobre triángulos y las nebulosas estelares de fondo se crean sobre un objeto sky_sphere).

Como recordaréis, los discos de «POV» se crean con la sentencia disc del lenguaje escénico. Estos discos son infinitamente delgados y pueden tener opcionalmente un agujero en su centro. Colefax utiliza estos objetos para representar estrellas, galaxias y otros objetos, y el resultado es de un sorprendente realismo. Además, los discos se generan más rápidamente que los halos por lo que el método permite también al usuario ganar velocidad con respecto al método descrito más arriba.

Colefax ha resuelto también fácilmente el problema de la posible superposición de objetos estelares con los modelos de primer plano (posibles naves espaciales). La solución de Colefax ha sido "pegar" todos los objetos estelares (estrellas, cometas, etc.) en la pared interna de una esfera gigante imaginaria (obteniendo así un modelo del universo muy parecido al de aquella antigua idea que sostenía que el firmamento era una bóveda rígida en la que

las estrellas estaban colgadas). De esta manera el volumen interior de la esfera imaginaria –o esfera-toldo– puede reservarse para colocar los modelos creados por el usuario sin peligro de que estos interfieran con los objetos celestes. Naturalmente este volumen interior puede resultarnos algo pequeño en ciertos casos y por ello Colefax nos ofrece la posibilidad de ampliarlo alterando el valor de una variable.

Aunque teóricamente podemos crear objetos de cualquier tamaño en «POV», en la práctica pueden darse cuelgues del programa al intentar trabajar con objetos muy grandes. Además, la velocidad del render puede ralentizarse a medida que aumenta el volumen espacial a tratar. Por esta razón los objetos creados por este POV-IPAS son relativamente pequeños y puede darse el caso de que una de nuestras naves espaciales tenga unas dimensiones superiores a las de las estrellas y galaxias creadas por el paquete.

Pero veamos qué posibilidades nos ofrece este POV-IPAS. El paquete está formado por cuatro ficheros: galaxy.inc, galaxy.bg, galaxy.obj y galaxy.sf. Y el sistema que conforman está diseñado para ofrecer diferentes posibilidades de empleo al usuario.

La primera opción (y la más simple) es limitarse a invocar al archivo galaxy.inc, el cual incluirá a los ficheros restantes y pasará valores a algunas de sus variables para crear un paisaje estelar; cuyo aspecto dependerá del valor de semilla que hayamos especificado con la variable galaxy_seed.

La siguiente posibilidad es realizar cambios manuales sobre los resultados obtenidos con un paisaje aleatorio, y la última es desechar el uso de galaxy.inc y crear un paisaje estelar a medida. Para ello el usuario habrá de decidir primero qué objetos desea incluir en su fondo estelar y luego experimentará con diversas variables antes de invocar a alguno de los restantes ficheros del paquete (o a todos).

Aquí tenéis algunos ejemplos de los fondos que pueden generarse con Galaxy.inc.

La opción más simple

Podemos generar nuestro primer fondo estelar simplemente incluyendo las dos líneas siguientes en nuestro fichero escénico:

```
#declare galaxy_seed = 123456789
#include "GALAXY.INC"
```

La variable galaxy_seed es la semilla para el generador de números pseudoaleatorios. Cada valor diferente dado a la semilla producirá un fondo estelar diferente. Galaxy.inc presupone que el centro de la esfera-toldo está en las coordenadas <0,0,0> y también supone que la cámara se halla situada en la misma posición y que está enfocada mirando a lo largo del eje Z, en la dirección negativa de éste. Es necesario recordar esto porque, por defecto, este archivo genera los objetos celestes más interesantes (los racimos de estrellas, las nebulosas cercanas, los cometas, etc.) teniendo esto en cuenta. Así, si hacemos rotar a la cámara para que apunte en otra dirección, seguiremos viendo un fondo estelar, pero la escena resultante será mucho menos espectacular.

Ahora comencemos a hacer experimentos con algunas de las variables del paquete. Tanto el funcionamiento de Galaxy.inc como el del resto de los archivos del paquete puede ser controlado dando valores a ciertos variables de control. Naturalmente estos valores han de ser escritos antes de la sentencia include que carga el fichero correspondiente donde las variables producen su efecto. Veamos por ejemplo cómo cambiar las dimensiones de la esfera-toldo:

```
#declare galaxy_distance=20000
```

Nuestro primer experimento (en pruebas.pov).

En nuestro segundo experimento añadimos un racimo de estrellas cercanas.

Aquí manipulamos a las estrellas del plano de fondo.

Trasteando con las variables de la nebulosa.

La línea precedente dobla el valor que por defecto galaxy.inc asigna al diámetro de la esfera-toldo pero, si incluimos esta línea en el ejemplo precedente, el fondo estelar que obtendremos será el mismo. Esto se debe a que galaxy.inc volverá a escalar todos los objetos estelares para que conserven el tamaño relativo asignado por defecto. El único cambio estará en que ahora dispondremos de un volumen espacial mucho mayor para situar nuestras naves (o lo que sea) en la escena, puesto que el diámetro de la esfera se ha doblado. Sin embargo, hay que tener cuidado con esto, debido a la posible lentitud del render, de modo que quizá nos resulte más práctico empequeñecer la escala de nuestros modelos en lugar de aumentar el tamaño de la esfera.

En cuanto a las demás variables de galaxy.inc, podemos comentar que Galaxy.colour se emplea para alterar las intensidades de color RGB de la escena. Estas variaciones no son absolutas sino que se efectúan sobre los valores de color que el generador ha asignado ya a la escena. Veamos un ejemplo:

```
#declare galaxy_colour= <1.5, 1, 0.5>
```

Esta línea hace que aumente la intensidad del componente rojo de la escena, deja el verde sin cambiar y reduce la fuerza del azul. (Nota: Colefax emplea realmente un doble juego de variables con el fin de soportar la versión inglesa y la americana de algunas palabras. Así, por ejemplo, podemos emplear galaxy_color en lugar de la variable del ejemplo o galaxy_coloration en lugar de galaxy_colouration).

Probablemente nos resultará difícil anticipar los resultados del uso de galaxy_color pero afortunadamente Colefax ha preparado otras variables cuyo manejo nos resultará más intuitivo. Una de ellas es galaxy_coloration, que sirve para incrementar o reducir la intensidad del color del fondo estelar res-

petando el tono básico. Así, la línea...

```
#declare galaxy_coloration=1.2
```

...intensifica el tono del fondo, sea cual sea, produciendo una escena con colores más vivos. Los valores inferiores a 1 producen el resultado opuesto.

Otra variable muy interesante es galaxy_intensity, que nos servirá para aumentar o reducir el grado de transparencia de la nebulosa estelar con respecto al fondo de estrellas que se encuentra detrás de dicha nebulosa. Esta variable requiere, como la anterior, un valor en flotante y su valor por defecto es de 1. Aquí tenéis un ejemplo:

```
#declare galaxy_intensity=1.5
```

Esta línea aumenta la opacidad de la nebulosa. También puede resultarnos interesante la posibilidad de mover el fondo estelar sin cambiar la cámara (que de esta manera seguirá enfocando a las posibles naves que existan en el primer plano). Esto podremos lograrlo utilizando la variable galaxy_rotate a la que daremos un vector donde se indicarán los grados de rotación para cada eje. Así, con la línea siguiente...

```
#declare galaxy_rotate= <30,90,0>
```

...el fondo estelar rotará 30 grados en el eje X y 90 en el Y.

Los otros ficheros

El archivo galaxy.inc no hace sino dar valores a ciertas variables de otros ficheros del paquete antes de incluir a éstos para crear las escenas. Podemos pues prescindir de galaxy.inc, asignar directamente los valores precisos a las variables convenientes e incluir directamente a uno o a todos los ficheros restantes del paquete en nuestro archivo de escena. Veamos qué hacen estos archivos.

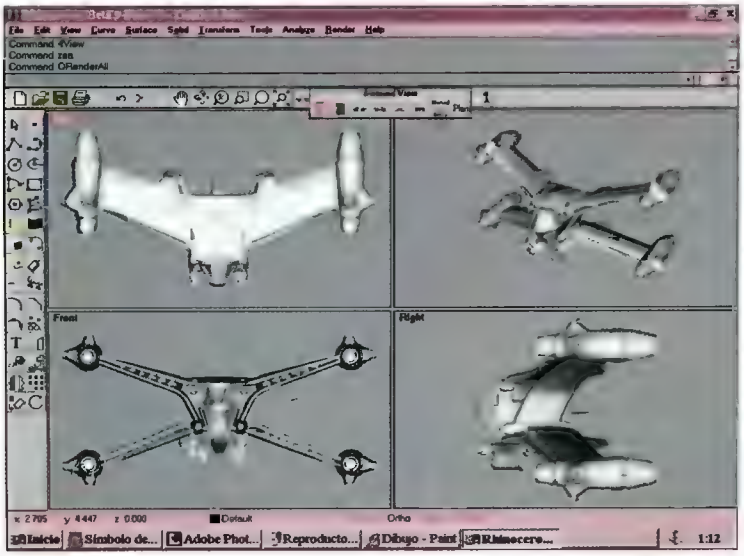
Galaxy.bg está encargado de crear la nebulosa estelar de fondo que suele formar parte de la mayoría de las escenas que generaremos. Galaxy.sf se emplea para crear la capa de estrellas de tamaño mediano (galaxy.bg añade también estrellas que se reducen a simples puntos para aparentar una mayor lejanía). Por último, galaxy.obj es el fichero encargado de crear los objetos estelares situados en primer plano. (Nota: los objetos de galaxy.obj serán los que perderemos de vista si cambiamos la orientación de la cámara).

Ahora vamos a comenzar a trastear con estos archivos, pero realizaremos nuestros experimentos sobre escenas completas generadas con galaxy.inc, con el objetivo de apreciar mejor cómo afectan nuestras pruebas al conjunto. Para ello podéis echar un vistazo al fichero pruebas.pov, donde hemos guardado una pequeña lista de pruebas. Para generar alguna de estas pruebas, eliminad los caracteres "/*" y "*/" que definen el bloque de líneas de cada experimento como comentarios. Para probar otro experimento volved a marcar el bloque anterior como comentario –para anular su procesamiento por parte de POV– y quitad las marcas-comentario del experimento que deseéis probar. Por supuesto, también podéis añadir aquí vuestros propios experimentos. Estudiemos el caso del primer experimento:

```
("experimento 1")

#declare galaxy_seed = 8563461
#declare options_only = true
#declare debug_options=true
#include "GALAXY.INC"

// Experimento num. 1
#declare galaxy_object_name="Star1"
#include "GALAXY.OBJ"
#include "GALAXY.BG"
#include "GALAXY.SF"
        ...
        ...
        ...
```



Preparando la conversión a POV del fabuloso caza modelado por Mark Kane.

En este experimento la línea que asigna el valor true a la variable options_only es muy importante. Al incluirla, el usuario le estará pidiendo a galaxy.inc que anule la inclusión de los ficheros galaxy.obj, galaxy.bg y galaxy.sf, pero todo lo demás seguirá funcionando igual. Esto quiere decir que galaxy.inc preparara las variables necesarias para los demás ficheros para crear una escena aleatoria, pero tendrá que ser el usuario quien escriba los includes necesarios para cargar a galaxy.obj, galaxy.bg y galaxy.sf.

Esto deja abierta la posibilidad de que cambiemos algunas de las variables definidas por galaxy.inc para estos archivos, lo que tendrá como consecuencia ciertos cambios con respecto a la escena final que galaxy.inc generaría con la semilla que le hayamos asignado. Esto convierte a options_only en una de las variables más importantes creadas por Colefax, puesto que su uso nos permitirá comprobar cómo afecta cada cambio a una escena válida generada por el paquete, como deseábamos hacer.

En cuanto a debug_options, al asignarle el valor true estaremos pidiendo a galaxy.inc que imprima en pantalla los valores asignados a las variables de construcción del fondo estelar. Con ello podremos tomar nota de las variables que se van a tratar con un valor determinado de semilla y podremos crear nuestras modificaciones mas fácilmente.

Nota: Colefax utiliza la sentencia #debug del lenguaje escénico de «POV» para imprimir textos en pantalla y los valores de sus variables.

Experimentos

Para el primer experimento nos hemos limitado a añadir una estrella del tipo 1. Pero esta estrella no es la única protagonista de la imagen puesto que también aparecen cuatro meteoros en el centro de la misma. Estos meteoros han sido creados por galaxy.inc, que siempre intenta crear unos objetos del tipo "primer plano" para realzar las escenas. La colocación de estos objetos puede llevarse a cabo de dos maneras: podemos emplear la variable galaxy_object_name, con la que podremos situar uno de estos objetos en la imagen, o bien podemos utilizar la variable galaxy_cluster_name, con la que se dibujará un grupo de cuerpos celestes (cluster=grupo o racimo).

En el ejemplo, galaxy.inc ha utilizado galaxy_cluster_name a la que ha asignado el nombre "Meteor1" mientras que nosotros hemos utilizado galaxy_object_name con el nombre "Star1". Estos nombres se han tomado de la lista de objetos celestes de primer plano que ha creado Colefax. En esta lista el nombre para cada tipo de objeto se forma añadiendo un número al nombre de la clase a la que pertenece. La familia de las estrellas se llama "Star" y consta de cuatro tipos diferentes de estrellas (es decir, que los miembros son Star1, Star2, Star3 y Star4). También hay una clase de nebulosas ("Nebula") que tiene seis miembros, otra de galaxias ("galaxy") que consta de cinco elementos, otra de cometas ("Comet") que tiene tres miembros y una de meteoros ("Meteor") de la que hay dos tipos.

Todos los objetos de "primer plano" se colocan por defecto en el centro de la escena, lo que puede acabar recargando la imagen si no hacemos algo. Podríamos, por ejemplo, eliminar a los meteoros de la escena.

Para ello sólo tendríamos que incluir la línea "#declare galaxy_cluster_name=". Esto le diría a galaxy.obj que no hay ningún grupo de objetos de primer plano en la escena. (El mismo truco vale para galaxy_object_name).

También podríamos crear varios objetos sueltos o varios grupos de ellos pero alterando su colocación en la escena. Esto podemos hacerlo utilizando la variable galaxy_object_position que se emplea para cambiar la colocación de los objetos individuales y de los grupos de objetos (cluster) en la esfera-toldo. La variable requiere la asignación de un vector en el que se emplean los valores dados en grados a los ejes X e Y para desplazar al objeto a cualquier nueva po-

sición en la esfera-toldo (el componente de Z es ignorado). Esto significa que si aplicamos al experimento anterior la siguiente línea...

```
#declare galaxy_object_position=<0,180,0>
```

...tanto la estrella como los meteoros desaparecerán de la escena y tendremos que hacer rotar la cámara 180 grados en X o Y para volver a verlos.

Otra variable a tener en cuenta es galaxy_object_rotate. Esta variable se emplea para hacer girar los objetos a lo largo del eje de visión Z y por tanto no necesita un vector, sino sólo un valor en flotante para indicar los grados deseados. La emplearemos para que los objetos celestes no queden demasiado alineados en la escena. Para comprobar bien su funcionamiento lo mejor es emplear un objeto grande y bien definido, como alguna galaxia.

El aspecto de los objetos

Aunque el acabado de los cuerpos celestes diseñados por Colefax es fantástico, los resultados pueden no ser tan buenos con ciertos colores o empleando una escala mayor. Esto es fácil de apreciar haciendo pruebas con la variable galaxy_object_scale. Esta variable sólo necesita un valor en flotante ya que el cambio de escala es uniforme en todos los ejes (un valor de 2 dobla el tamaño del objeto mientras que otro de 0.5 lo reduce a la mitad). Si escalamos un objeto como Galaxy1 el resultado será bastante bueno, incluso aunque hagamos al objeto tres veces más grande, pero esta misma prueba resultará desastrosa utilizando Galaxy5, que, sin embargo, queda bien con la escala 1 empleada por defecto.

Otra variable que podemos emplear para cambiar el acabado de los objetos es galaxy_object_flatten, a la que daremos un valor en flotante que servirá para aplanar o ensanchar la forma de los objetos. Para comprobar cómo funciona tendréis que renderizar una escena con el objeto Galaxy1 y luego volver a realizar la prueba añadiendo la línea:

```
#declare galaxy_object_flatten=0.8
```

Esto hará que la galaxia parezca estar más inclinada con respecto al espectador. El valor dado por defecto es 0.2. Con un valor de 0 no hay aplanamiento mientras que con otro de 1 el objeto quedará totalmente aplanado. Finalmente, otro punto a tener en cuenta es que no todos los objetos quedan bien como grupos. Esto resultará visualmente adecuado en el caso de estrellas y meteoros, pero no en el de galaxias y nebulosas.

Grupos de objetos de primer plano

Ahora tomad nota de las variables que controlan a los objetos cluster. En primer lugar tenemos a galaxy_cluster_objects, a la que asignaremos un valor con el número de los objetos que van a formar parte del grupo a crear. También podemos usar galaxy_cluster_scale (que requiere un valor en flotante) para aumentar o disminuir el tamaño de los objetos individuales que componen al cluster. Otra variable interesante es galaxy_scale_turb, a la que daremos un valor en flotante para aumentar o disminuir las diferencias de tamaño entre los objetos que componen el cluster. Asignando un valor de 0, todos los objetos del cluster tendrán el mismo tamaño mientras que con 1 las diferencias de tamaño serán bastante extremas (el valor dado por defecto a

esta variable es de 0.5). Colefax ha pensado incluso en una variable para evitar la excesiva simetría en los brillos de los grupos de estrellas. En estos casos podremos emplear a galaxy_rotate_turb para variar aleatoriamente la rotación (en el eje Z) de las estrellas del grupo. Para ello asignaremos a esta variable un valor en flotante que oscilará de 0 a 360, consiguiendo en este último caso una variación aleatoria de entre 0 y 360 grados para cada una de las estrellas del grupo. Nota: creemos que en este caso la uniformidad es mejor.

La llamarada de los cohetes está hecha con un par de halos.

Por último, puede interesarnos cambiar las distancias entre los componentes del grupo. En este caso emplearemos la variable galaxy_cluster_spread a la que asignaremos un valor en grados que variará de 0 a 360. El valor dado por defecto es de 15. Con un valor de 360 los componentes del grupo se esparcirán por todo el cielo. Nota: en el experimento 2 tenéis pruebas de todas estas variables.

El fondo estelar

Tampoco debemos descuidar el campo estelar de medio fondo. Tomad nota del uso de las siguientes variables (experimento 3):

star_count: Es el contador de estrellas.

star_colour: Es el color de las estrellas. El valor RGB por defecto es <0.9, 0.9, 0.9>.

star_type: Usaremos el valor 1, 2 ó 3 para elegir el tipo de las estrellas.

star_scale: Esta variable sirve para fijar el tamaño medio de las estrellas de fondo. El valor dado por defecto es de 1 pero probablemente tendremos que cambiarlo dependiendo de la resolución con la que estemos trabajando. (Un valor de 0.5 reduce a la mitad el tamaño medio y un valor de 2 lo dobla).

star_distance: realmente no hay una sola esfera-toldo sino dos: una para la nebulosa de fondo y otra para el fondo estelar. El diámetro de esta última se fija con star_distance que por defecto está a 20000.

La nebulosa

Por último, aquí tenéis unas notas sobre algunas de las variables que controlan la nebulosa de fondo:

galaxy_bgnebula: con esta opción indicaremos, utilizando un valor entre 1 y 6, el tipo de pigmento que deseamos para la nebulosa de fondo. Si no se desea una nebulosa de fondo hay que poner esta variable a "false".

galaxy_nebula_ambient: emplea un vector (que por defecto es <1,1,1>) para cambiar el valor de la luz ambiente de la nebulosa.

galaxy_colour1, 2 y 3: únicamente necesitaremos dar un valor para galaxy_colour1 si queremos crear una nebulosa de un solo color. Con las otras dos variables podremos añadir dos colores base más pero el resultado puede ser bastante confuso.

galaxy_patter_scale: esta variable, que por defecto está al valor 1, nos servirá para escalar la trama de la nebulosa sin cambiar su forma básica.

Notas:

•Existen tres variables que se emplean para desactivar la creación de los objetos que generan los ficheros galaxy.bg, galaxy.sf y galaxy.obj.

Estas variables son, respectivamente, galaxy_bg, galaxy_starfield y galaxy_objects. Para eliminar de la escena a los objetos que crean los archivos a los que se refieren, basta con ponerlas a false. Estas variables son puestas a false automáticamente siempre que se incluye un galaxy.inc, por tanto habrá que ponerlas a true antes de invocar a galaxy.inc otra vez dentro del archivo escénico.

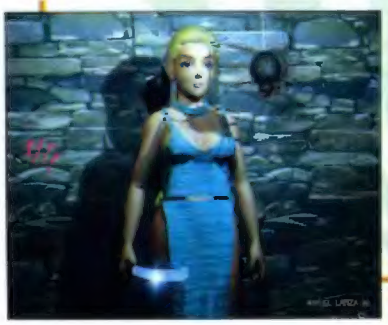
•Aún falta por añadir un último detalle, la iluminación. Tenéis que recordar añadir las luces necesarias para iluminar a los modelos propios que añadáis a la escena. Si no lo hacéis así vuestros objetos sólo serán sombras recortadas contra el fondo estelar.

Nota importante. Podéis remitirnos vuestros trabajos o consultas, bien por carta a la dirección que figura en el sumario de Pcmanía, o vía e-mail a rendermania.pcmania@hobbypress.es

Poesía en imágenes

Una de las ventajas que posee la publicación, en un mismo espacio, de un conjunto de imágenes es la variedad. Una variedad que afecta no sólo a los programas empleados para su realización o a los temas tratados, sino también al propio estilo que impregnan las imágenes. En efecto, las obras que incluimos en esta sección desprenden fantasía, sentido del humor e incluso poesía.

Quienes contemplen las escenas de este autor ya se habrán dado cuenta que la palabra poesía se refería a la obra de **Ángel Lanza Salcines**. Ángel, quizá algo abrumado por los elogios (merecidos) que le dedicamos en el número 14, asegura que aún tiene que progresar mucho. Es posible, desde luego, pero –con independencia de tu nivel técnico (que no nos parece nada desdeñable)– creemos que tus escenas son extraordinarias por la magia y belleza que transmiten. Por nuestra parte te agradecemos tus elogios pero lo que más nos ha emocionado es la escena Jmm.jpg, donde Pamela posa con algunos de nuestros viejos modelos. ¡Gracias! En cuanto a tus opiniones sobre el arte en general y la infografía en particular nos han parecido interesantes y queremos animar al resto de los lectores a que envíen sus propias opiniones sobre estos y otros temas. Para dar ejemplo ahí va la nuestra: cualquier soporte de información puede ser un medio válido para crear arte. El arte, la magia, la poesía, pueden estar en una escena sintética, en una partida de ajedrez, en un programa, en un libro, en un cómic e incluso en una demostración matemática. Pero todo dependerá de lo que el artista consiga plasmar en su obra. Por otra parte, creemos que muchos de los trabajos que se aceptan hoy como arte no son más que tomaduras de pelo o especulaciones comerciales. En fin, en esta ocasión tenemos dos cartas de Ángel (la segunda está en el directorio salcines\otra). En una de ellas las escenas están montadas en una página HTML y en la otra tenemos una resultona animación en la que aparece otra de las impresionantes chicas de Ángel y de la que se adjunta un divertido "making of".

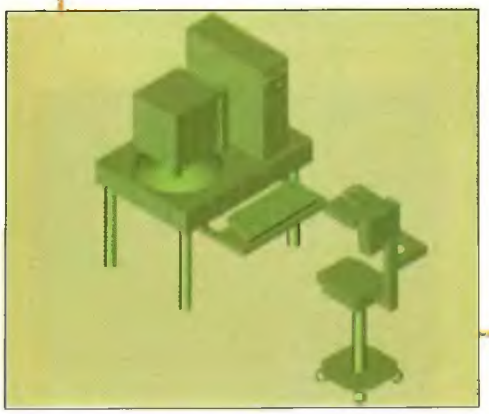


El Foro del Lector

José Luis García Fuertes es un povmaníaco a la vieja usanza. Los modelos que nos envía han sido creados con el lenguaje escénico de «POV» y renderizados con él. Deseando participar en la escena de Mechs, Jose Luis nos envía un bonito y original aparato parcialmente articulable desde «POV». Si has leído los últimos números de Rendermanía ya habrás visto que no es tan difícil crear piernas y brazos articulables.

No te preocupes, tu modelo aparecerá en la próxima portada de Mechs y lo mismo podemos decirte del precioso submarino que has preparado para los orcos, que intervendrá en la próxima batalla donde éstos participen. Es decir, lo incluiremos si nos envías el modelo, porque te has olvidado de incluirlo en el disquete. Por cierto, el diseño del submarino es muy gracioso pero ¿se puede abrir la escotilla trasera para que el comandante asome la cabezota? Y en ese caso, ¿la escotilla no debería estar en otro sitio para que la torreta no obstruya la visión?

Julián Hortolano Villarejo nos envía un escritorio virtual y tiene varias dudas que intentaremos aclarar: «POV» es un programa de orientación distinta a la de «Spatch». Este último es un modelador que sólo dispone de un pequeño render de trabajo mientras que «POV» tiene un render de gran calidad, pero no una interfaz gráfica para crear modelos. Por otra parte, para crear escenas de calidad tendrás que exportar tu modelo a «POV» y renderizarlo desde este programa. Esto puedes hacerlo de varias maneras: exportando el modelo directamente a «POV» –en cuyo caso éste se grabará como patches bicúbicos (un objeto de «POV»)–, o bien grabarlo como DXF y reconvertir este formato con alguna utilidad para que «POV» pueda leerlo. La lentitud de redibujado de «Spacth» se debe a que los puntos que maneja son puntos de control de splines. Las formas generadas con splines requieren más cálculos que las típicas formas poligonales, pero si te lo montas bien emplearás muy pocos puntos. (Nosotros pondríamos más RAM en tu ordenador aunque probablemente esto no afectará a la velocidad de redibujado de «Spatch»). Y sí, habrías podido hacer la misma escena con bastantes menos puntos de control, pero este es un defecto que cometemos todos alguna vez.

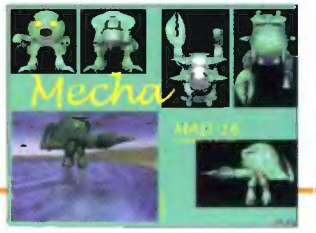


Luis Martín Manguero nos pide una crítica de las imágenes que ha preparado utilizando «3D Studio». Bien, tus escenas tienen una buena iluminación aunque las sombras te han quedado, quizá, demasiado difusas (en el caso del despertador). En una de las entradas de la sala de baños la textura de ladrillos se te ha corrido (deberías haber dividido la aplicación). Además, sucede algo raro en estas entradas. La escena que más nos ha gustado ha sido la del despertador, por su sentido del humor.

José Antonio Aroca nos envía varias escenas espaciales creadas con «Caligary Truespace». Entre ellas nuestra favorita es "objetivo" que está sacada de una viñeta de Tintín y que nos parece la más conseguida (aunque es una pena que no hayas podido emplear los halos de «POV» para la llamarada del cohete). Esperamos volver a verte pronto por aquí.

Alberto y **Oscar Otero Leal** nos envían una divertida animación realizada con «MAX» en la que intervienen un búho y una rata. Según cuenta Oscar, esperaban conseguir un look parecido al de las tiras de algunos periódicos y... lo han logrado. Los modelos de los personajes son sencillos, pero esto se debe al propio diseño planeado en papel para los mismos y no a una falta de conocimientos. En su carta Oscar nos cuenta los pormenores de la construcción de ambos modelos e incluye detalles acerca de los sonidos que han puesto a la animación. Enhorabuena a ambos por vuestro trabajo.

Alain S. Bañuls Vilela es otro povmaníaco que nos envía un par de mechs que aparecerán en la próxima portada sobre mechas. Alain también nos envía su último trabajo: una reconstrucción de la Punta de la Barca (Muxia. La Coruña). Esta escena ha sido construida con «POV», aunque el autor también se ha apoyado en «Spatch» (hemos eliminado los ficheros a petición del autor), y todavía no está completa del todo ya que Alain aún está luchando con las rocas de la playa. Echamos en falta algún pov-ipas que cree escenarios de terreno para planos cercanos de rocas, desfiladeros, playas, etc. Nos han gustado mucho tus mechas. El anfibio de desembarco tiene una forma muy original, parece un cangrejo, y el otro es muy gracioso. ¿Qué programa has empleado para hacerlos? ¿Y por qué no los has preparado para que se articulen desde« POV»? Gracias por tu contribución a la artillería pesada mecha.



Gonzalo Blázquez ha conseguido crear una escena desértica de gran realismo utilizando «3D Studio» y algunos programas de retoque fotográficos. Y es que este autor ha puesto "a mano" detalles como los reflejos del sol en el cristal delantero y las sombras (así como el soldado del fondo). En definitiva, una gran escena llena de detalles con texturas bien aplicadas y un modelo estupendo a pesar de su relativa simplicidad. Pero; ¿por qué la escena tiene tanto granulado? Y por favor, la próxima vez envía la carta como fichero.

El Foro del Lector

Roberto Font Ruiz nos envía varias escenas y animaciones que están muy bien teniendo en cuenta la poca experiencia infográfica de este autor. Entre ellas figura una animación creada con la idea de responder al reto del combate de magos que proponía Juan B. Garraza Zurbano. En dicha animación un mago-caballero camina y lanza una bola energética. ¿Cómo arbitrar éste u otros futuros combates? Lo ideal sería que ambos contendientes apareciesen en la misma animación, pero esto sería bastante difícil por razones obvias. Por tanto proponemos lo siguiente:

1) Un autor publica una escena o animación acerca de mechs, brujos, chicas o lo que sea y lanza un reto.
2) Las animaciones de las que hablamos aquí no deberán exceder los 5 Megas comprimidas.
3) Alguien recoge el guante publicando la respuesta con una escena o animación, dependiendo de lo que haya exigido el retador, de su propio modelo.
4) Los lectores disponen de tres meses, a partir del momento en que se publica la respuesta, para votar en el foro.
5) El ganador tendrá derecho a publicar una escena o animación de victoria mientras que el perdedor, si es un buen deportista, estará obligado a enviar una escena o animación en la que se ilustre su derrota. Estos trabajos deberán enviarse antes de que se cumplan cuatro meses después de la publicación del fallo.
6) El perdedor tendrá la posibilidad de obtener la revancha, entendiéndose que el combate se producirá contra el último trabajo del mismo tipo que haya publicado mientras tanto el retador. Esto no evitará al perdedor el cumplimiento de la cláusula anterior. En este caso la votación podrá empezar con la publicación del nuevo trabajo del derrotado.
7) La recogida del guante supone la aceptación de todas las condiciones anteriores so pena de escarnio público.

Naturalmente, estos combates pueden ser múltiples si más de un lector recoge el guante en un mismo número de Rendermanía. Además, si la idea tiene éxito podríamos mejorarla... ¿Qué decís? ¿Quién se anima?

En cuanto al trabajo de Roberto, está hecho con «3D Studio» y «MAX». Las escenas no están mal aunque nos han gustado más las animaciones. De tus escenas, la espacial cuyos fondos has generado con «MAX» es nuestra preferida. ¿Qué opinas de los fondos que crea el Pov-ipas de Colefax?

Francisco José Serrano Rey ha sufrido lo suyo para utilizar un paisaje de HF-lab con «3D Studio 4», por lo que la lectura de su carta puede ahorrar sufrimientos inútiles a quienes planeen hacer algo parecido. Francisco deseaba un buen paisaje de fondo para la escena donde exhibe su Mech Marauder. Pero lo cierto es que la posición de la cámara, lógicamente centrada en el Marauder, no permite captar demasiados detalles del fondo montañoso. En cuanto al Mech de la escena, es una buena recreación del Mad-3R Marauder de Battletech. Bien, Francisco, en nombre de todos: gracias por tus notas sobre Hf-lab.